MEMORY ACCESS BUFFERING IN MULTIPROCESSORS†

Michel Dubois, Christoph Scheurich, Faye Briggs*

Computer Research Institute
University of Southern California
Los Angeles, California

*Dept. of Electrical and Computer Eng.
Rice University
Houston, Texas

ABSTRACT

In highly-pipelined machines, instructions and data are prefetched and buffered in both the processor and the cache. This is done to reduce the average memory access latency and to take advantage of memory interleaving. Lock-up free caches are designed to avoid processor blocking on a cache miss. Write buffers are often included in a pipelined machine to avoid processor waiting on writes. In a shared memory multiprocessor, there are more advantages in buffering memory requests, since each memory access has to traverse the memory- processor interconnection and has to compete with memory requests issued by different processors. Buffering, however, can cause logical problems in multiprocessors. These problems are aggravated if each processor has a private memory in which shared writable data may be present, such as in a cache-based system or in a system with a distributed global memory. In this paper, we analyze the benefits and problems associated with the buffering of memory requests in shared memory multiprocessors. We show that the logical problem of buffering is directly related to the problem of synchronization. A simple model is presented to evaluate the performance improvement resulting from buffering.

1. INTRODUCTION

Shared memory MIMD systems are becoming very popular because of their versatility and the fact that they are a natural evolution from traditional architectures based on the proven von Neumann single-processor concept. Off-the-shelf, low-cost microprocessor components can be connected in a tightly-coupled configuration [ARC85], and attain computing speed comparable to that of single-processor mainframes. Similarly, in high-end machines, the limit to technology improvements leads to the use of tightly-coupled configurations to meet the ever-increasing demand for computing power. This trend is clear from observing the design of recent machines such as the IBM308X and the Cray-XMP [HWA84a]. At the same time, many advanced designs for shared memory multiprocessor supercomputers [HWA84b] are being researched both in academia and in industry: examples are the Cedar at the University of Illinois, the NYU Ultracomputer, and the IBM RP3 multiprocessors.

Tightly-coupled multiprocessor systems with a limited number of very powerful pipelined processors are possible computer architectures for achieving the billions of instructions per second that will be required from the data processing systems of the next decade. In such architectures, it is important to keep the processor efficiency high because of the large cost of each processor. High utilization of processors is guaranteed by a steady, uninterrupted supply of instructions and operands. Pipelining and buffering of memory accesses must be used in order to bridge the gap between the slower shared memory and an execution unit with a very short cycle time.

In a multiprocessor, processor interconnections to shared memories are more complex than the dedicated memory bus found in single processors. Thus, transfers between processors and memories are slower. The effects of memory access conflicts and latency delays can be reduced by connecting a local memory to each processor. As a cache, the local memory is allocated dynamically at run-time and addressed associatively. If the local memory is part of the physical address space of a process, then it is allocated at compile- or load-time and addressed randomly [PHI83]. Local memories may contain shared data. In local caches which allow shared data, the problem of multiple, inconsistent copies of the same data in different caches appears [CEN78, DUB82]. This problem is often solved by hardware - in general, a processor that writes into its cache sends invalidation signals to all the other caches possessing a copy of the block which contains the modified word. If a random access local memory contains shared data, the multiple copy problem does not exist, but the local memories must be addressable by all processors. In the Cm*, for example, operand accesses to the global memory are routed automatically - i.e., through hardware - to the computer module containing the operand.

Three different multiprocessor systems are studied in this paper from the point of view of memory access buffering. These structures are representative of the system structures for shared memory multiprocessors.

- **System 1: Shared global memory system**
  A multiprocessor with shared global memory and private, local memories. The shared memory is accessed by all processors. Local memories are associated with specific processors and do not contain any shared data (see Figure 1).

- **System 2: Distributed global memory system**
  A multiprocessor with distributed global memory which consists of an interconnection of local memories. The local memories contain global, as well as private data and are accessed randomly (no multiple copies of data exist) (see Figure 2).

† This research is supported by an NSF Research Initiation Grant No DMC-8505328 and the USC Faculty Innovation Fund

- System 3: Cache-based system

  A multiprocessor with shared global memory and private caches. The shared memory contains code and data, and the caches are accessed associatively. The caches may or may not contain shared data. We will consider only the cases of write-through and write-back caches (see Figure 3).

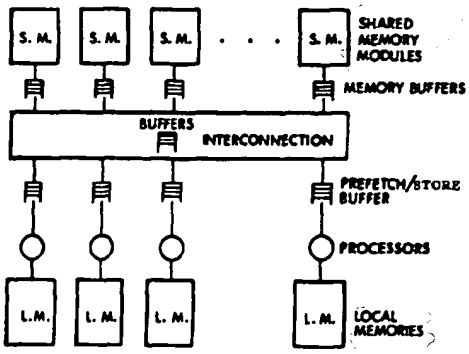

Figure 1: System with a shared global memory.

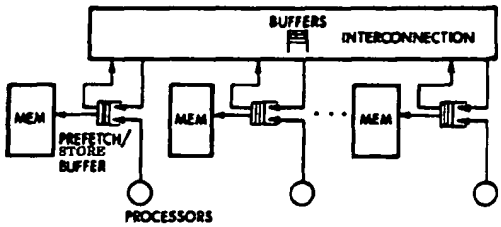

Figure 2a: System with distributed global memory. Remote accesses are buffered in PREFETCH/STORE buffers.

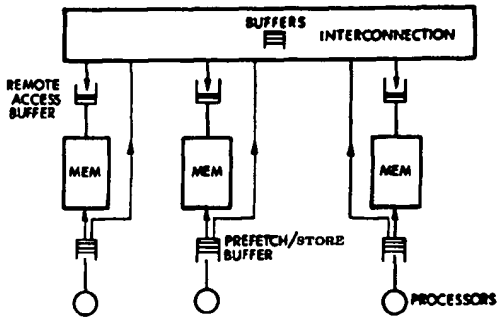

Figure 2b: System with distributed global memory. Remote accesses are buffered in a buffer independent of the PREFETCH/STORE buffers.

In these three systems, we will study the effects of buffering of memory requests on the coherence of the multiprocessor and on its performance. The buffering of memory accesses has been shown to improve the performance of pipelined processors in many studies. Buffers can be included in the processor, the cache, the interconnection or at the memory. The performance advantages of buffering stem from the improvement in the overall system throughput through the pipelining of execution and memory accesses.

In section 2, we review memory access buffering techniques. In a single processor system, the only logical problems limiting buffering are the dependencies between successive instructions. In a shared memory multiprocessor, logical problems also arise because of the concurrent environment. In section 3, these problems are reviewed and the notions of strong and weak coherence are introduced and discussed in the context of system 1. In section 4 and 5 possible buffering techniques are identified for systems 2 and 3 in the contexts of weak and strong coherence. Finally, in section 6, we propose extensions to the concepts and results presented in the paper.

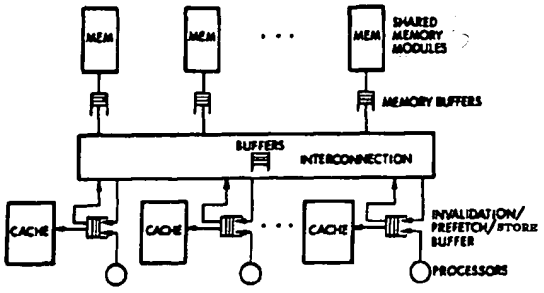

Figure 3a: Cache-based system. Invalidations are buffered in the PREFETCH/STORE buffers.

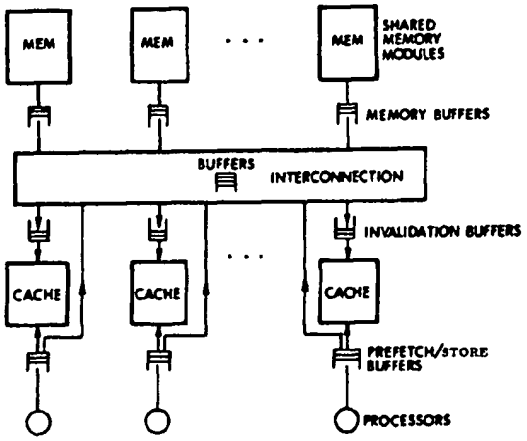

Figure 3b: Cache-based system. Invalidations are buffered in independent INVALIDATION buffers.

2. MEMORY ACCESS BUFFERING TECHNIQUES

In high-end, general-purpose processors, the pipelining of instruction execution is common place. Extensive prefetching and buffering of memory accesses are implemented in pipelined computers. Prefetching anticipates data LOADs by the execution unit by early fetching and decoding of future instructions; moreover, buffering STOREs reduces a processor's idle time when waiting for an acknowledgement after a STORE operation.

Prefetching of instructions is one of the most common examples and is very effective because of the predictability of instruction sequencing. Operands are also prefetched in high-end machines. Often, an instruction can be decoded, its operand addresses computed, and the accesses to

memory initiated, long before the instruction is actually executed. Several operand fetches may be in progress at any time. Ideally, by the time the instruction enters the execution unit, all of its memory operands will have been fetched.

A scheme supporting operand prefetching is implemented in the IBM3033, by the operand registers [KOG81]. Up to six operand prefetches may be in progress at any time. The instruction decode unit initiates the prefetch by storing the effective address in one of six operand address registers. The memory requests are tagged so that, when the data arrives in the processor, it is stored in the corresponding operand data register, packaged with the execution unit. A similar procedure is implemented for the STOREs. STOREs to memory occur later in the pipeline, when the execution of an instruction is completed. Usually, STOREs can be completed in one pipeline cycle, by simply writing data and addresses in an operand STORE buffer. Up to four STOREs can be buffered in the IBM3033 machine. STORE buffers are particularly appealing because the processor does not need to wait for the return of information from memory, contrary to the LOAD of an operand. As a result, LOADs are more critical for performance and are often given higher priority over STORE requests.

To support a processor with extensive operand buffering, a cache is usually pipelined and may be lockup-free [KRO81]. A lockup-free cache does not block (or lock up) the processor on a miss. Rather, it records the status of the memory request causing the miss and keeps accepting requests from the processor. When the cache locks up the processor, the processor must stop issuing memory requests. If the cache locks up memory accesses on each miss and resolves one miss at a time, the prefetching mechanism of the processor will not be effective. Since memory interleaving across cache blocks [BRI83] can speedup the resolution of several concurrent misses, lockup free caches may significantly decrease the average miss penalty. In [KRO81], a lockup-free cache is described in which several misses can be buffered in the cache and can be resolved concurrently by the cache controller.

In a multiprocessor system, memory accesses can be buffered in the processor, in the interconnection network, and at the shared memory. Buffering at the shared memory has been analyzed in [BRI79]. In some designs, several paths may be possible between a processor and a memory module [CHI84]. If invalidations or STOREs have to modify an entry in a local memory of another processor, these updates may be buffered in the destination processor node and they may have lower priority than the accesses issued by the processor directly connected to the local memory. A good example is the BIAS filter in the IBM3033 to store invalidation signals coming from other caches [SMI82]. Unless they are buffered, the accesses coming from other processors will reduce the bandwidth between the processor and its local memory.

Buffering instruction fetches is safe for a pipelined machine in a multiprocessor if it can be assumed that instructions are not modifiable, We will not discuss instruction fetches any further. The following sections address more specifically the problems associated with operand fetching and storing.

Another trivial case in which buffering is not a problem is throughput-oriented multiprocessor systems, in which the processors execute completely independent processes. In this case, there is no constraint imposed on buffering because of the multiprocessor environment. The discussion in this paper applies to multiprocessor systems in which several processes sharing data can be scheduled on different processors at the same time. This is the case for multitasked systems and for some multiprocessor operating systems.

Buffer management refers to the order in which multiple buffered requests are treated. In most cases, the requests are treated in a strict FIFO order (First-In-First-Out). In some cases requests may be allowed to pass each other in the buffer. This is referred to as *jockeying*. Jockeying is often permitted between memory requests for different memory words, but is not permitted between requests destined to the same memory word. Jockeying with this restriction is called *restricted jockeying* in the rest of the paper.

3. SEQUENTIAL CONSISTENCY AND COHERENCE IN MULTIPROCESSORS

A simple uniprocessor generally executes instructions one at a time, in the order specified by the program. If the processor is pipelined however, then several consecutive instructions may be executed concurrently or even out of their intended order. This is allowable in uniprocessors, provided hardware mechanisms (interlocks) exist to check data and control dependencies between instructions to be executed concurrently [KOG81]. This checking is local to the processor and can be done efficiently.

If processors are part of a multiprocessor which executes a concurrent program, then such local dependency checking is still necessary but not sufficient to preserve the correct outcome of a concurrent execution. Since data are shared and interrupts can be sent between processors, processes running on different processors may affect the outcome of each other. Enforcing data dependencies between processors which are physically distant is not as efficient as enforcing them in a single processor.

A strong requirement for the functional behavior of a multiprocessor system is *sequential consistency*. Lamport [LAM79] defines sequential consistency as follows.

Definition 3.1: Sequential consistency

> "[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

The definition above applies to systems where complete statements are not considered atomic. Atomicity is limited to LOADs and STOREs. For example, in a statement such as $C \leftarrow A + B$, where A, B, and C are in memory, events from different processors may be interleaved between the LOADs of A and B and the STORE of C. There is an implicit ordering of events, e.g., A is first fetched then B is fetched, then C is stored.

The only way that two concurrent processes can affect each other's execution is through the sharing of data and the sending of interprocessor interrupt signals. Take for example the multitasked program of Figure 4a. In this program, two processes synchronize to access a critical section through shared variables A and B. We define a *legal interleaving* of memory accesses an interleaving such that the references from each process appear in program order. The

```
                PROCESSOR 1

        A:=0
         .
         .
        A:=1                    /event S1(A)
LAB1:   If (B=1) GOTO LAB1      /event L1(B)
        <critical section>
        A:=0


                PROCESSOR 2

        B:=0
         .
         .
        B:=1                    /event S2(B)
LAB2:   If (A=1) GOTO LAB2      /event L2(A)
        <critical section>
        B:=0
```

Figure 4a: Synchronization protocol using two shared variables A and B.

legal interleavings of executions of events on shared variables A and B in program 4a are displayed in Figure 4b. All 6 legal interleavings of the first two statements of each program are possible. In some cases legal interleavings may be impossible because they correspond to the sequences that the programmer wants to avoid. Some of the possible interleavings result in deadlocked sequences. Note that, in Figure 4a, if processor 1 is allowed to prefetch B before setting A to 1, and if processor 2 is allowed to prefetch A before setting B to 1, an illegal interleaving may result in which both processors enter the critical section at the same time. The interleaving is illegal because the events on shared data in the two processors do not appear in their intended logical order in the interleaving.

Legal Interleavings:	Result:
S1(A)->L1(B)->S2(B)->L2(B)	Processor 1 enters CS.
S2(B)->L2(A)->S1(A)->L1(B)	Processor 2 enters CS.
S1(A)->S2(B)->L1(A)->L2(B)	Deadlock.
S1(A)->S2(B)->L2(B)->L1(A)	Deadlock.
S2(B)->S1(A)->L1(A)->L2(B)	Deadlock.
S2(B)->S1(A)->L2(B)->L1(a)	Deadlock.

Figure 4b: Legal Interleavings of STORE and LOAD events.

Since the user considers only legal interleavings when a program is written, a sequentially consistent multiprocessor must only allow legal interleavings of events. The only data dependencies to resolve between two different processes are on *shared data* (we ignore interrupts). The coherence enforced on shared data by the memory system is therefore very relevant to this study. Censier and Feautrier define a coherent memory scheme as follows [CEN78].

Definition 3.2: Memory system coherence

> "A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address."

In an environment where STOREs can be buffered in a STORE buffer associated with each processor, the notion of *latest value* is vague. It is not clear whether "latest STORE" refers to the execution of the STORE by a processor, or to the update of memory. In order to refine the definition of memory system coherence, we differentiate between *initiating, issuing* and *performing* a memory access.

* The processor environment includes the CPU and local buffers.

Definition 3.3: Memory request initiating, issuing, and performing

> A request is *initiated* when a processor has sent the request and the completion of the request is out of its control. An initiated request is *issued* when it has left the processor environment* and is in transit in the memory system. A LOAD by processor I is considered *performed* with respect to processor K at a point in time when the issuing of a STORE to the same address by processor K cannot affect the value returned to processor I. A STORE by processor I is considered *performed* with respect to processor K, at a point in time when an issued LOAD to the same address by processor K returns the value defined by the STORE.† An access by processor I is *performed* when it is performed with respect to all processors.

Because of dependencies within each instruction stream, the definition implies that an access by processor K is performed with respect to processor K as soon as it is initiated.

For example, in system 1 (Figure 1), operand prefetching is implemented through a prefetch buffer at the processor. The processor *initiates* the operand prefetch by placing the address in that buffer. Then the buffer controller *issues* the operand fetch to the shared interconnection and memory. The request transits in the interconnection and it is *performed* when it is latched in a buffer associated with the memory, provided this buffer is FIFO (restricted jockeying is allowed in the memory buffer). The situation is similar for a STORE request. The processor *initiates* the STORE by placing the request in a STORE buffer at the processor. Later on, the STORE buffer controller *issues* the STORE request to the interconnection. The request is then in transit in the interconnection. It will be *performed* as soon as it is latched in the FIFO buffer associated with the destination memory.

We want to emphasize at this point the basic difference, in general, between *issuing* and *performing* a memory access. A memory STORE cannot affect *any* other process before it is issued (and similarly, a memory LOAD cannot be affected by *any* other process until it is issued). When the STORE is *issued but not performed,* it *may* affect the issuing of a LOAD of the same data by any processor; at the time when the STORE is *performed* it *is certain* that it will affect the issuing of a LOAD on the same data by another processor.† The distinction between *issued* and *performed* is important for the analysis of cache-based systems and systems with recombining interconnection networks such as the network proposed for the NYU Ultracomputer [HWA84b].

From definition 3.3, it is clear that definition 3.2 of memory system coherence refers to *performed* STOREs. Collier [COL84, COL85] has extensively studied the problems of coherence and event ordering in a multiprocessor system where each processor has its own copy of the global memory. He proves that a sufficient and (for all practical purposes) a necessary condition for sequential consistency is that all processors must "observe" STORE events in the same order. However, Collier does not consider operand prefetching. In the following definition, it is considered that a STORE on a

† Naturally, the issued LOAD is only affected by the particular STORE, if it is performed before a subsequent STORE is performed at the same address.

variable is *observed* by a processor at the time when the processor performs a LOAD on that variable which returns the value defined by the STORE.

Definition 3.4: Strong ordering of storage accesses

In a multiprocessor system, storage accesses are *strongly ordered* if
1) accesses to global data by any one processor are initiated, issued and performed in program order, and if
2) at the time when a STORE on global data by processor I is observed by processor K, all accesses to global data performed with respect to I before the issuing of the STORE†† must be performed with respect to K.

It follows from logical considerations on the timing of events in a multiprocessor [LAM79], that a coherent system with strong ordering of events is sequentially consistent. Condition (1) constrains the ordering of accesses on global data to be in program order. The only way that a processor I can affect another processor K is by I modifying a global variable, X, and by K subsequently reading the value. Condition (2) guarantees that all global accesses issued and observed by I before the issuance of the STORE request "happened before" all global accesses issued and observed by K after the LOAD request is performed [LAM79].

Condition (1) is necessary as demonstrated by the example in Figure 4. Condition (2) is essential if there are more than two processors in the multiprocessor, as the following example demonstrates. Refer to Figure 5a and b. In the following, Si(X) and Li(X) represent global accesses "STORE by processor i in X" and "LOAD of X by processor i, respectively. If L2(A) reads the value produced by S1(A), and if L3(B) returns the value produced by S2(B), then L3(A) must also read the value produced by statement S1(A). However, if, for some reason, event S1(A) takes much more time to propagate to processor P2 than it does to processor P3, then there may be enough time (depending on conflicts and distances) for P2 to perform event L2(A), then S2(B), and for P3 to perform L3(B) before S1(A) has been performed. This may result in P3 performing L3(A) on the value of A previous to S1(A), and in an illegal interleaving of events. Note that the problem comes from the fact that S1(A) was performed with respect to P2 when P2 initiated S2(B); it should therefore be performed with respect to P3 when L3(B) is performed. Of course, the possibility of such an occurrence depends greatly on actual machine timing, but this example shows how difficult it is to design cache-based systems and systems with recombining interconnection network in which events are strongly ordered.

Processor 1	Processor 2	Processor 3
.	.	.
.	.	.
S1(A)	L2(A)	L3(B)
.	S2(B)	L3(A)
.	.	.
.	.	.

Figure 5a: Three processes sharing variables A and B.

The complexity of definition 3.4 comes mainly from the fact that STOREs can be observed at different times by different processors before they are performed. If STOREs can only be observed once they are performed, then condition (1) is sufficient. In this case the STOREs are "atomic" and therefore, as soon as a global access is performed with respect to any processor K, it is performed with respect to all processors. System 1 behaves in this manner and we now examine the requirements imposed by strong ordering on the design of system 1.

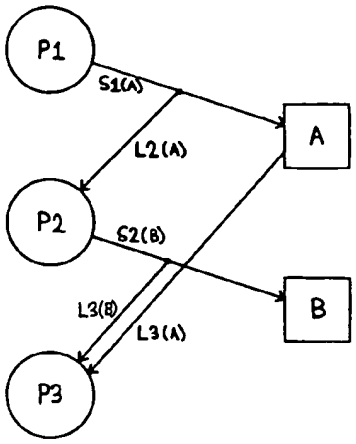

Figure 5b: Possible outcome of concurrent execution of programs in Figure 5a.

Example: Strong ordering of events in system 1

This system has been analyzed by Lamport [LAM79]. We first look at the requirements to satisfy condition (1) of definition 3.4. If the delay through the network is constant or bounded for all requests (a rare case in practice because of conflicts), or if there is only one path from processors to memories (e.g., in single bus systems) then successive requests can be issued in program order without waiting for acknowledgements from the memory. In general, however, because of conflicts, the only way that a processor can ensure that its global data requests are performed in program order is to issue the requests one at a time and to wait for an acknowledgement after each request. Since the STOREs are atomic, condition (1) is sufficient. The following are the rules for enforcing strong ordering of events in system 1.

1) Global memory accesses can only be performed at the memory.

2) Individual processors initiate global data accesses in program order. These accesses (both for LOADs and STOREs) are buffered in the same local buffer associated with the processor. Therefore, the STORE and PREFETCH buffers of a pipelined machine must be logically unique. The combined buffer is managed by a strict FIFO policy. Internal forwarding [KOG81] (i.e., bypassing the memory) in a processor is restricted by condition (1).

3) The controller of the combined PREFETCH/STORE buffer issues and performs the memory accesses one-by-one, in the FIFO order of the buffer.

The first requirement precludes recombining interconnection networks, in which a LOAD request "colliding" with a STORE request for the same data in a switch box is combined so that the STORE is sent to memory and the LOAD returns the value defined by the STORE. In such a system,

†† It is assumed here that the value to STORE is known at the time of issuance. In [LAM79], it is considered that the value could be defined after the STORE has been performed.

guaranteeing condition (2) and avoiding the problem described in Figure 5 is difficult in general.

To maintain strong ordering, dependencies on *every* data access to shared memory have to be checked. However, most of these data are not *synchronizing variables*, i.e., shared variables used to control the concurrency between several processes (such as variables A and B in Figure 4). In multitasked programs such variables are used to synchronize processes and to maintain the integrity of shared modifiable data structures or variables.

We can identify several sources of inefficiencies in the strong ordering algorithm given above. First, restricted jockeying should be allowed in the PREFETCH/STORE buffer associated with the processor: a processor does not need the result of STORE references, but LOAD references are particularly critical because of local dependencies in the pipeline. Therefore, performance can be improved if LOADs are allowed to pass STOREs in the combined PREFETCH/STORE buffer (provided the LOAD and the STORE are not for the same address). If the interconnection network is complex and packet-switched, performance is also improved if the controller of the PREFETCH/STORE buffer does not have to wait until each shared memory access is performed before issuing the next memory access, and can empty the buffer at the faster issue rate, rather than the perform rate. Also, to speed-up the CPU, the PREFETCH and the STORE buffers can be separate altogether. The policy in which shared memory accesses can be issued optimally is called *weak ordering of events* and is introduced and defined below.

In a system with a weak ordering of events, two types of shared variables are distinguished: first the shared operands appearing in algorithms whose value does not control the concurrent execution; and second synchronizing variables which protect the access to shared writable operands or implement synchronization between different processes. If a shared variable is modified by one process and appears in other processes and, if the access to the variable must be protected, then it is the responsibility of the programmer to ensure mutual exclusion for each access to the variable by using high-level language constructs such as critical sections [AND83]. Critical sections are in turn implemented by basic synchronization primitives such as locks. It is assumed that, at run time, the system can distinguish between accesses to synchronizing variables and to other shared variables. Synchronizing variables can be distinguished by the type of instruction (TEST_AND_SET, COMPARE_AND_SWAP, RESET, FETCH_AND_EXECUTE, or special LOAD and STORE instructions, for example).

Definition 3.5: Weak ordering of events

> In a multiprocessor system, storage accesses are *weakly ordered* if
> 1) accesses to global synchronizing variables are strongly ordered and if
> 2) no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and if
> 3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

The dependency conditions on shared variables are weaker in such a system, because they are only limited to hardware-recognized synchronizing variables. Between operations on such variables, no assumption can be made by the programmer of a process on the order in which STOREs are propagated and observed. The order of successive STOREs by a processor, to the same address, is however respected. Buffering and restricted jockeying are allowed in all buffers, except for operations on hardware-recognized synchronizing variables.

In order that the program of Figure 4 executes correctly in a system with a weakly ordering of events, variables A and B must have been declared as synchronizing variables. Special LOAD and STORE instructions may therefore be generated by the compiler for such variables. It is interesting, that in the concurrent language ADA [AND83], such synchronizing variables may be specified in a pragma (i.e., an advice to the compiler). This provision was most probably included to prevent harmful optimizations by the compiler for such variables.

A weakly ordered system is not sequentially consistent. If the compiler is capable of detecting shared variables used for synchronization (such as A or B in Figure 4), it could generate special LOAD and STORE cycles for such variables. We feel that such detection may however be difficult without an explicit declaration from the programmer.

Simple performance analysis of buffering in system 1

A schematic representation of a shared memory multiprocessor system is given in Figure 6. Processors execute locally until an access to shared memory has to be performed. An access request traverses the processor/memory interconnection, and is serviced by the appropriate bank of the shared, interleaved memory. The state of each processor alternates between compute and wait phases. A processor "computes" while it accesses local data. On a reference to shared data, it may have to wait while the shared data is not available. Let t_p be the average duration of the compute phase between two successive data accesses to shared memory in one of the P processors. If p_s is the probability that an instruction contains an access to shared data and $I_{s.p.}$ is the MIPS rate of a processor when all accesses are local (single processor configuration), then $t_p = 1/(I_{s.p.}\ p_s)$. The interconnection is characterized by t_{issue} and $t_{perform}$, the minimum times to issue and to perform a request respectively. There are M memory modules, which are all accessed with equal probability, and the access time of a memory bank is T_m. We will assume that the memory is sufficiently interleaved, so that it is never a bottleneck. For example, M can be large enough, so that $P/t_p < M/T_m$. We also neglect all dependencies in the CPU and conflicts to access shared memory (i.e., the memory system has enough bandwidth.). The following results show the relative effect of the two buffering strategies. Let t_{interf} be the average interreference time between two consecutive accesses to shared variables by the same processor, i.e., it is the total duration of the compute and the wait phases between two accesses to shared memory; we have:

$t_{interf} \geq t_p + t_{perform} + T_m$ (no buffering at the processor),

$t_{interf} \geq MAX\left[t_p, t_{perform}\right]$ (buffering with strong ordering), or

$t_{interf} \geq MAX\left[t_p, t_{issue}\right]$ (buffering with weak ordering).

In the first equation T_m may be equal to zero if buffers are implemented at the memory. In the second and third results, we have assumed buffering at the memory. The first inequality results from the fact that, in a non-buffered system, the processor is blocked every time it performs a shared memory access. In the second or third cases, the processor and the buffer controller form a pipeline with average segment times of t_p and $t_{perform}$, or t_p and t_{issue}, respectively. The throughput of this pipeline is determined by the bottleneck segment.

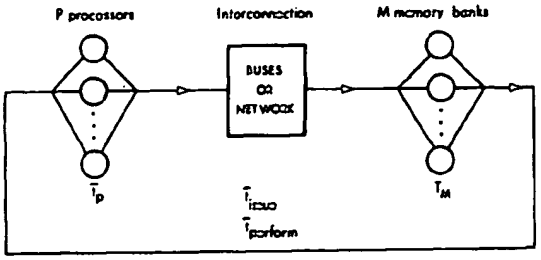

Figure 6: Schematic representation of a multiprocessor system and its shared memory.

The efficiency of the multiprocessor system denoted E is the ratio of the MIPS rates of a processor in the tightly-coupled ($I_{t.c.}$) and in the single processor ($I_{s.p.}$) configurations.

$$E = (I_{t.c.} / I_{s.p.}) = (t_p / t_{inref}).$$

From the above formulas, we can see that the effectiveness of buffering shared data accesses depends on the relative values of t_p and t_{issue} or of t_p and $t_{perform}$. Note that the value of t_p depends both on the MIPS rate of the processor as a single processor, and on the probability that an instruction references data in shared memory. In Figure 7 the multiprocessor efficiencies in the cases of no buffering at the processor and of buffering with strong ordering are compared as a function of $t_{perform}/t_p$. Buffering is more attractive for highly-pipelined machines, and for cases where the access rate to shared memory is high. In Figure 8, we have assumed that $t_{issue} < t_p$ and the multiprocessor efficiency of buffered systems with weak and strong ordering are compared as

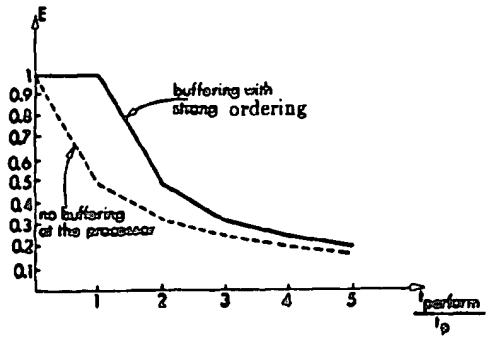

Figure 7: Comparison of systems without buffering (dashed line) and with buffering and strong ordering (solid line).

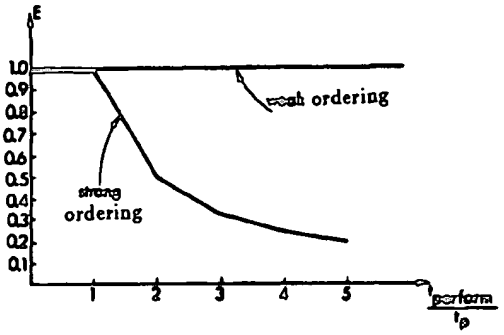

Figure 8: Comparison of buffering with strong and weak ordering ($t_{issue} < t_p$).

$t_{perform}/t_p$ increases. This would be the case if the number of processors increases and the interconnection network is packet-switched; so that the delay through the network increases, but the time to issue remains the same. It appears from the simplified model that weak ordering in buffered system is only effective for systems where $t_p < t_{perform}$. While the simple models give indications, they must be interpreted with caution since many effects have not been taken into account (conflicts, dependencies, finite buffer length, synchronization, jockeying in buffers...) A more extensive model would be justified, but it is beyond the scope of this paper.

4. MULTIPROCESSORS WITH DISTRIBUTED GLOBAL MEMORY

In the architecture of system 2, each processor has a local memory. Private and shared data can be placed in that local memory. Each memory is accessible by all processors. We assume that, at the hardware level, the distinction between global and private data cannot be made. Logically, this system is equivalent to system 1 with no private memory, but the shared memory access time is not uniform; it depends on whether the access is local or remote. In system 2, all STOREs are atomic. Condition (1) of definition 3.4 is therefore sufficient for strong ordering and sequential consistency.

A well-known example of system 2 is the Cm*, built at Carnegie-Mellon University. In the Cm*, a STORE or a LOAD cycle must be fully acknowledged before the processor can proceed with its execution. The access times of references in the local memory, in the memory of a processor of the same cluster, or in a memory of a remote cluster are 3, 9 and 26 microseconds, respectively. During that time a processor is blocked. Since all accesses are acknowledged, and since the LSI-11s (the processors used in the Cm*) are non-pipelined and remain blocked during each access, the Cm* is strongly ordered. If buffering is implemented at the processor, then strong or weak coherence are possible alternatives.

In Figure 2, two organizations of the buffers are presented. In the first case (Figure 2a), the remote accesses are buffered at the destination in the same buffer as the local accesses of the destination processor. As in system 1, the buffer management is strictly FIFO. A remote memory access is issued in this case when it is sent to the interconnection network. A local access is issued and performed as soon as it is placed in the local buffer. A remote access is per-

formed when it is placed in the buffer of the remote memory. With these definitions, the conditions for strong and weak ordering are similar to the conditions for system 1, and the performance models are the same, except that the probability of a shared data reference in an instruction must be replaced by the probability of a remote data reference.

In the second case (Figure 2b), remote accesses are buffered in a distinct remote access buffer. To maximize processor/local memory bandwidth, remote accesses may be given a different (lower) level of priority. To enforce strong ordering in this system, local or remote accesses should be considered *performed* only once they are executed in the local or the remote memories, respectively. The time to perform an access in this system may therefore be quite long. Strong ordering may be inefficient. Weak ordering is preferable. In the case of weak ordering, the STORE/PREFETCH buffer issues local references by starting the memory cycle and issues remote references by simply latching them in the first stage of the interconnection. However, whenever a processor executes an instruction on a declared synchronizing variable (this is detected by the fact that the data have been tagged by the compiler, or that special instructions are used), it must ensure that all its previous data accesses have been performed, and stop issuing PREFETCHes and STOREs of operands until the access to the synchronizing variable is also performed.

5. CACHE-BASED MULTIPROCESSORS

In a cache-based multiprocessor, each processor has a local cache and the cache contains data and instructions from shared memory. We distinguish between the cases of software- and hardware-enforced coherence. Caches may be write-through or write-back caches [SMI82].

5.1 Software-enforced coherence

If no shared writable data can be loaded into cache then no coherence problem exists between the caches. This technique relies on software to avoid the coherence problem. At any time, the caches contain private data or non-modifiable shared data. The distinction is done at compile-time, possibly with some indication from the programmer. Accesses for shared writable data in the shared memory can be buffered at the processor in a common PREFETCH/STORE buffer. With respect to buffering, the problems with cache-based systems described in this paragraph are very similar to the problems analyzed in system 1. Other cache systems with software enforced coherence are possible but are not considered here.

5.2 Hardware-enforced coherence

Of particular interest is the case of cache-based systems with hardware-enforced coherence [CEN78, DUB82]. We only discuss data caches that can contain shared data. Instruction fetches are not part of this discussion. If the caches contain shared writable data, coherence between multiple copies of these data is maintained through hardware invalidation signals. Also, in some coherence algorithms, a processor may broadcast a LOAD to all caches and to the memory in the case of a miss, in order to read the data directly from another cache. Algorithms to enforce coherence abound in the literature. Analyzing in details the implications of strong and weak coherence on cache coherence protocols will be the topic of a future paper. In this paper, we simply introduce some alternatives. The problem of event ordering is much more complex for cache-based systems than it is for the previous two systems. For example, while a STORE and its resulting invalidations are in progress, copies of the modified variable may exist in different caches as well as in the shared memory and therefore a STORE may not be atomic. In order to enforce strong ordering of events, we briefly discuss how to implement conditions (1) and (2) of definition 3.4 for the two system configurations shown in Figure 3. In the following, P-data refers to data that are private to the cache (one single copy exists) and S-data refers to data that are shared among several caches (several copies may exist in different caches)[DUB82].

Two buffer configurations are shown in Figure 3. In Figure 3.a, there is a unique buffer per processor. The buffer contains data PREFETCH and STORE requests for the local processor plus the accesses made by remote processors (LOADs or INVALIDATEs). The local processor initiates its STOREs and LOADs in the local PREFETCH/STORE buffer. No jockeying is allowed in this buffer. The local buffer controller issues requests to the cache one at a time. A STORE request on S-data in the cache will require invalidating the data in other caches. A STORE issued (Figure 3a) by processor I is performed with respect to processor K when the cache is updated (hit on P-data) or at a point in time when it is placed in the memory buffer and when an invalidation (if it is necessary) has been placed in the local buffer of processor K (access to S-data). There are several ways to enforce condition (2).

In the first solution, the LOADs causing misses and the STOREs on S-data causing INVALIDATEs are broadcast to all caches and to memory so that they are performed with respect to all processors at the same time. A cache can read a missing block from a different cache. It is therefore assumed that a LOAD issued by processor I is performed with respect to processor K after the cache cycle (hit) or after the request has been placed in the memory buffer and in the local buffer of processor K (miss). This means that LOADs and STOREs can be performed atomically; this solution can be applied easily to systems with a few buses [ARC85].

Another solution satisfying condition (2) is inspired from the paper by Collier [COL85]. In this case, the caches could be connected by a point-to-point interconnection such as a ring or a mesh. When a STORE on S-data must propagate invalidations, the location in shared memory is first locked to prevent any processor from accessing it. The invalidation is then sent to each cache, one by one but always in the same order, by propagating invalidations through the point-to-point interconnection. When the invalidation has been placed in the local buffer of each cache, the STORE is performed in memory. At this time the STORE is considered performed, and the processor issuing the STORE may issue the next global memory access. A LOAD missing in the cache for which a STORE is in progress will be rejected at the shared memory because the STORE has locked the memory block. It should be retried (alternatively the LOAD could be delayed at the memory in a special buffer). The LOAD is only performed when it is accepted by the memory (i.e., after the STORE has released the memory block). It can be easily shown that such a scheme satisfies condition (2) of definition 3.4. This comes from the facts that LOADs are performed atomically in the cache or at the memory, that the STOREs on P-data are performed atomically at the cache, and that the STOREs on S-data (i.e., causing INVALIDATEs) are performed one after the other and in ord-

er. We believe that the second solution is good for non-bus cache-based systems, with a centralized directory [CEN78, DUB82] in which broadcasting is impracticle.

In the system of Figure 3b, the buffer for INVALIDATEs and for LOADs issued by remote processors is distinct from the buffer for LOADs and STOREs from the local processor. An invalidation of a block in a remote cache is performed when it is executed in the cache. It is difficult to maintain atomicity of STOREs (first solution above) because the invalidation buffer of each cache may contain different numbers of invalidation requests, making the time to invalidate each cache random. The second solution is possible and will enforce strong coherence. The only difference is that an invalidation signal must invalidate a cache before moving to the next cache (in the system of Figure 3.a, an invalidation is simply placed in the local buffer).

In a weakly ordered system, the processors can issue shared memory request without waiting for previous requests to be performed. This would result in a system with very high efficiency. In this case, the only troublesome accesses are accesses to synchronizing variables. The buffer controller must still record the status of all cache accesses that it has issued but not performed, so that it can perform them every time an access to a synchronizing variable is detected. The implementation of such a buffer may be very complex. Deadlocks are also possible. The details of an implementation for a given cache coherence mechanism would be interesting in order to understand the practical aspects of the concept of weak ordering in cache-based systems, but it is beyond the scope of this paper.

6. CONCLUSION

We have presented in this paper a framework to analyze the coherence properties of shared memory multiprocessor systems when data accesses are buffered at the processor and in the interconnection between the processors and the shared memory. The concepts and results presented in this paper are extensions of Lamport's results [LAM79]. We have introduced three states in which a shared memory request may be. We have demonstrated that these states are fundamental by using them to define the notion of strong ordering when data accesses are buffered and by showing the equivalence between strong ordering and sequential consistency. To alleviate the performance problems with strong ordering, we have introduced the concept of weak ordering of events. Weak ordering results in the highest possible processor efficiency. A weakly ordered system is not sequentially consistent. The programmer must declare explicitly what we have called synchronizing variables, i.e., variables used to synchronize processors, and to protect the integrity of shared writable data through mutual exclusion.

Three systems were analyzed in this study under no buffering, buffering with strong ordering and buffering with weak ordering. In the case of system 1 a simple model was presented to identify the range of system parameters under which the three policies are effective. The model could be extended to other cases and more sophisticated models are warranted to fully highlight the advantages of buffering in specific systems. The fundamental approach taken in this paper has allowed us to identify the basic restrictions on buffering imposed by the two ordering policies in the case of some very complex systems, such as cache-based systems. We believe that more work is warranted in this direction.

7. REFERENCES

[AND83] G.R. Andrews, *et al.*, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Vol.15, No. 1, March 1983.

[ARC85] Special session on commercial cache-based multiprocessors, in the *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985.

[BRI79] F.A. Briggs, "Effects of Buffered Memory Requests in Multiprocessor Systems," *Proceedings of the Conference on Simulation, Measurements, and Modeling of Computer Systems*, 1979.

[BRI83] F.A. Briggs and M. Dubois, "Effectiveness of Private Caches in Multiprocessors with Parallel-Pipelined Memories," *IEEE Transactions on Computers*, January 1983.

[CEN78] L. M. Censier and P. Feautrier,"A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, Vol. C-27, No.12, December 1978.

[CHI84] C-Y Chin and K. Hwang,"Packet-switching Networks for Multiprocessor and Data-flow Computers," *Proceedings of the 11th International Symposium on Computer Architecture*, June 1984.

[COL84] W. W. Collier, "Architectures for Systems of Parallel Processes," *IBM Technical Report* TR00.3253, January 27, 1984.

[COL85] W. W. Collier, "Reasoning about Parallel Architectures," submitted to *JACM*, 1985.

[DUB82] M. Dubois and F.A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers*, Vol. C-31, No. 11, November 1982.

[GEH82] E.F. Gehringer, *et al.*, "The Cm* Testbed," *IEEE Computer*, October 1982.

[HWA84a] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, Mac Graw-Hill.

[HWA84b] *Tutorial on Supercomputers: Design and Applications*, Kai Hwang Ed., IEEE Computer Society, 1984.

[KOG81] P. M. Kogge, "The Architecture of Pipelined Computers," Mac Graw-Hill, 1981.

[KRO81] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th Annual Symposium on Computer Architecture*, June 1981.

[KUN76] H.T. Kung, "Synchronized and Asynchronous Parallel Algorithms for multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub Ed., New York: Academic Press, 1976.

[LAM78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. of the ACM*, July 1978, Vol.21, No. 7.

[LAM79] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. C-28, No. 9, September 1979.

[PHI83] L. Philipson, et al., "Communication Structure for a Multiprocessor Computer with Distributed Global Memory," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

[SMI82] A.J. Smith, "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, September 1982.